Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software

Andreas Zankl[1] (✉), Johann Heyszl[1], and Georg Sigl[2]

[1] Fraunhofer Research Institution AISEC, Munich, Germany
{andreas.zankl,johann.heyszl}@aisec.fraunhofer.de
[2] Technische Universität München, Munich, Germany
sigl@tum.de

Abstract. The shared instruction cache of modern processors is an established side-channel that allows adversaries to observe the execution flow of other applications. This has been shown to be a threat to cryptographic software whose execution flow depends on the processed secrets. Testing implementations for these dependencies, or *leaks*, is essential to develop protected cryptographic software. In this work, we present an automated testing methodology that allows to detect execution flow leaks in implementations of modular exponentiation, a key operation in schemes like RSA, ElGamal, and Diffie-Hellman. We propose a simple and effective leakage test that captures problematic properties of vulnerable exponentiation algorithms. The execution flow of an implementation is directly monitored during exponentiation using a dynamic binary instrumentation framework. This allows to efficiently detect leaking code with instruction-level granularity in a noiseless and controlled environment. As a practical demonstration, we test multiple RSA implementations of modern cryptographic libraries with the proposed methodology. It reliably detects leaking code in vulnerable implementations and also identifies leaks in a protected implementation that are non-trivial to spot in a code review. We present a fix for these leaks and strongly recommend to also patch the other implementations. Because instruction cache attacks have been shown to be a threat in practice, it seems advisable to integrate an automated leakage test in the software release process of cryptographic libraries.

Keywords: RSA, Granular Leakage Detection, Open-source Crypto Library, Non-trivial Bug Discovery, Noiseless Evaluation Environment, Automated Software Testing, DBI

1 Introduction

Using cryptographic algorithms securely in practice is challenging, because many implementations are vulnerable to the exploitation of side-channels. Execution

time, power consumption, electromagnetic radiation, fault behavior, and the micro-architecture of processors have all been used to attack implementations in practice. A subset of micro-architectural attacks focuses on the instruction cache, which is a high-speed memory close to the processing unit that is used to speed up code execution. Caches are typically transparent to applications, i.e. they work regardless of an application's awareness of the cache. Yet, an application can actively manipulate its cache usage. With sufficient details about the cache sub-system, code can actively be placed in a specific location in the instruction cache at a specific point in time. Because caches are also shared resources, an adversary can reconstruct an application's execution flow by cleverly replacing the application's code in the instruction cache and checking whether it has been reused after a while. This is a well-known threat to cryptographic software, because performance optimizations often shape the execution flow according to processed secrets. By observing the instruction cache, an adversary is then able to recover secret data or parts thereof. This spawned a number of instruction cache attacks that have been proposed over the past decade. As a consequence, execution flow dependencies that reveal, or *leak*, processed secrets should be removed from cryptographic software. In this work, we propose a testing methodology that allows to automatically detect execution flow leaks in implementations of modular exponentiation, a key operation in asymmetric ciphers such as RSA, digital signature schemes like DSA or ElGamal and key agreements protocols such as Diffie-Hellman.

In Section 2 we give an overview of related testing approaches and tools. We then briefly discuss common modular exponentiation algorithms and their general vulnerability to instruction cache attacks in Section 3. In short, vulnerable algorithms leak information about the exponent, because they optimize, e.g. skip, calculations depending on its value. These execution flow dependencies are the basis for instruction cache attacks. To automatically find them, we propose a simple and effective test in Section 4 that captures the optimization strategies of vulnerable algorithms such as square-and-multiply and sliding window exponentiation. The test assumes that leaking code is used more often, the more 1-bits occur in the exponent (or vice versa). Hence, the correlation of (1) the Hamming weight of the exponent and (2) the number of times an instruction is used during exponentiation systematically identifies vulnerable code. Testing this leakage model requires programs to be monitored during execution with instruction-level granularity. For this purpose we use the dynamic binary instrumentation framework Pin [13] known from software debugging and performance analysis. It is described in Section 5 and allows to take execution flow measurements in a controlled environment free from interferences from the processor, the operating system or other applications.

To practically demonstrate our methodology, we test the RSA implementations of nine open-source cryptographic libraries: BoringSSL [9], cryptlib [11], Libgcrypt [17], LibreSSL [21], MatrixSSL [12], mbed TLS [3], Nettle [19], OpenSSL [28], and wolfSSL [30]. The selection is briefly discussed in Section 6 and focuses on well known `C`-based cryptographic libraries. It is by no means ex-

haustive and does not follow any specific selection criteria other than the preferences of the authors. To verify our testing framework, we additionally test one self-written textbook implementation of RSA that is intentionally vulnerable to instruction cache attacks. Among the open-source libraries, not all use exponentiation algorithms that offer protection against instruction cache attacks. The test results discussed in Section 7 confirm this observation and, as a novelty, clearly state the extent of the leaks for each library. While MatrixSSL exhibits the most extensive leaks, the default implementations of OpenSSL and its forks as well as cryptlib do not leak at all. The tests also uncover leaks in the otherwise protected implementation of wolfSSL. In the analysis in Section 8 we show that these leaks are non-trivial to find and remove the responsible instructions with minor code changes that have negligible performance impact. In our conclusions in Section 9, we strongly recommend to fix all affected libraries and suggest to integrate automated leakage tests in the software release process of cryptographic libraries.[1] In summary, our main contributions are

- a simple and effective leakage test for implementations of modular exponentiation allowing to detect execution flow leaks that are the basis of instruction cache attacks,
- a testbench based on dynamic binary instrumentation that efficiently monitors the execution flow in a noiseless and controlled environment,
- a quantification and comparison of execution flow leaks in RSA implementations of a selection of modern cryptographic libraries, and
- an analysis and fix of leaks detected in the wolfSSL library demonstrating the necessity and benefits of the proposed leakage test.

2 Related Work

The detection of information leakage through a program's execution flow has been addressed in the context of static code analysis. Doychev et al. [7] as well as Doychev and Köpf [8] propose a static analysis tool called *CacheAudit*, which quantifies the amount of information leaked through cache observations. It takes a 32-bit x86 program binary and a cache configuration as inputs and reports the information leakage for different attacker models. Barthe et al. [4] propose a static analysis tool that checks whether implementations are constant-time, i.e. do not perform conditional jumps or memory accesses depending on secrets. The authors mainly evaluate symmetric ciphers with their tool, but also state that it successfully detects implementations of modular exponentiation that branch based on secrets. Rodrigues et al. [24] propose another static analysis tool called *FlowTracker*, which is directly incorporated into the compiler. With their tool, the authors detect not further specified time-based side-channels in the Montgomery exponentiation implementation of OpenSSL. In contrast to static analysis, we perform our leakage test based on runtime measurements for concrete

[1] To facilitate the integration, the source code used in this work can be obtained from `https://github.com/Fraunhofer-AISEC/cache-leak-detector`.

inputs. This allows us to specifically observe only active code paths, which reduces the evaluation effort for large and complex software components.

Basic runtime detection of execution flows leaks has been performed by Molnar et al. [20]. The authors propose the so-called *program counter security model*, in which a program's execution flow must be independent of any secret input such that an adversary does not gain any information from observing its program counter. While the focus of their work is put on program, i.e. source-code, transformations to achieve program counter security, they also analyze example programs to detect code that potentially violates program counter security. For these measurements, the authors use *gcov*, a code coverage tool from the GNU Compiler Collection (GCC). It instruments code compiled with GCC and reports the code coverage, e.g. how often a line of source code has been executed. If different inputs cause different numbers of executions, a potential program counter security violation is detected. There is no further notion of whether the number of executions are statistically related to the inputs. In contrast to the work by Molnar et al., our leakage test specifically identifies linear relations between the number of executions and the Hamming weight of the exponents in modular exponentiation implementations. In addition, our measurement framework has fewer restrictions regarding the software under test. Unlike *gcov*, it does not require access to the source code of an implementation. It is also not limited to code compiled with GCC and does not require additional compiler flags. Hence, it is easier to deploy in practice.

Langley [18] proposes a dynamic analysis tool called *ctgrind*, which is an extension to the dynamic binary instrumentation framework Valgrind. The tool allows to mark secret data as uninitialized and then uses the existing capabilities of the memory error detector to warn if a branch is taken or a memory lookup is performed depending on the uninitialized (secret) data. With the tool, Langley identifies secret-dependent memory accesses in a modular exponentiation implementation of OpenSSL. Reparaz et al. [23] analyze whether an implementation is constant-time by processing fixed and random inputs and measuring the corresponding execution times. The authors then use Welch's t-test to determine whether an implementation is constant-time or not. In contrast to both approaches, we propose a leakage test specifically constructed to capture vulnerabilities in modular exponentiation implementations. Furthermore, we systematically evaluate multiple modern cryptographic libraries as well as quantify and compare the extent of their execution flow leaks.

3 Modular Exponentiation and Instruction Cache Attacks

Modular exponentiation with large integers is a key computation in multiple cryptographic ciphers and schemes. On computer systems with limited native word length, special algorithms are needed to implement arbitrary length integer arithmetic. A common choice for modular exponentiation of the form $b^e \; mod \; p$ are so-called m-ary algorithms. Binary variants ($m = 2$) process the exponent e bitwise, from most to least significant bit or vice versa. The square-and-multiply

algorithm [15] executes a modular square operation for any bit e_l in the exponent, whereas a modular multiplication is performed only if $e_l = 1$. Obviously, this approach creates a highly irregular execution flow that directly depends on the bit values of the exponent. 2-ary algorithms with a more regular execution flow are the Montgomery ladder [14], which uses a square and a multiply step for any e_l, and the square-and-multiply-always algorithm [6], which simply adds dummy multiplication steps to balance the execution flow. Faster exponentiation with higher memory demand is achieved by fixed-window exponentiation (FWE) algorithms [10] ($m = 2^k$). They process the exponent in bit-windows of size k and use pre-computed powers of the current base $b^2, b^3, ..., b^{k-1}$. For every window, k modular square operations are performed together with an additional modular multiplication step using one of the pre-computed values. FWE algorithms exhibit a regular execution flow that is independent of the value of the exponent. Further optimizations are introduced in sliding-window exponentiation (SWE) algorithms [16], which for the same window size k require just half of the pre-computed values and fewer multiplications by skipping windows containing only zeros. Similar to the square-and-multiply algorithm this creates an irregular execution flow that again depends on the bit values of the exponent.

In the context of instruction cache attacks, which allow an adversary to observe the execution flow, traditional square-and-multiply as well as SWE implementations should be avoided, as previous literature shows. A practical instruction cache attack on square-and-multiply is shown by Yarom and Falkner [31]. With their attack the authors are able to recover over 90 % of the private exponent. Percival [22] and Acıiçmez [1] both demonstrate attacks on the SWE implementation used for fast RSA private key operations in the OpenSSL cryptographic library. By observing the exponent-dependent sequences of modular square and multiply operations, the authors are able to retrieve around 200 bits of each 512-bit Chinese remainder theorem (CRT) component used in the decryption. After the publication of Percival [22], OpenSSL refrained from using its SWE implementation for secret exponents [27]. Acıiçmez et al. [2] and Chen et al. [5] further improve the work by Acıiçmez [1] by relaxing instruction cache observation requirements and by proposing a better key recovery from SWE measurements.

Literature clearly shows that instruction cache observations can successfully be used to infer an application's execution flow. Because caches handle data in chunks, i.e. cache lines, adversaries can typically observe only groups of instructions that all fit on one cache line. The size of cache lines as well as other properties of the cache can vary in practice. To test software in worst case conditions, we assume an adversary is able to observe single instructions. If an implementation does not leak at instruction-level granularity, observations via the cache will not succeed, regardless of the cache configuration in practice.

4 Definition of the Leakage Test

Both square-and-multiply and sliding window exponentiation algorithms optimize the runtime by skipping calculation steps if there are 0-bits in the exponent.

When implemented in software, this causes some instructions, e.g. one function, to be executed more often, the more 1-bits the exponent contains. To systematically capture these execution flow variations in implementations of modular exponentiation, we correlate the Hamming weight (HW) of the exponent e with the number of times an instruction is executed during exponentiation. Note that the test also detects those instructions that are executed more often, the more 0-bits the exponent contains. This case would yield a negative correlation.

To test an implementation, N random bases are exponentiated with a corresponding number of random exponents $e_{n=1..N}$. This gives a sequence of Hamming weights $\mathbf{H} = \{\mathrm{HW}(e_1), ..., \mathrm{HW}(e_N)\}$. For all instructions i in the software, the number of executions $x_{i,n}$ is saved for each exponentiation. The sequences $\mathbf{X}_i = \{x_{i,1}, ..., x_{i,N}\}$ are then compared to $\mathbf{H}$ using Pearson's correlation coefficient, which yields

$$c_i = \frac{\sum_{n=1}^{N} \left(\mathbf{H}[n] - \overline{\mathbf{H}}\right) \cdot \left(\mathbf{X}_i[n] - \overline{\mathbf{X}}_i\right)}{\sqrt{\sum_{n=1}^{N} \left(\mathbf{H}[n] - \overline{\mathbf{H}}\right)^2} \cdot \sqrt{\sum_{n=1}^{N} \left(\mathbf{X}_i[n] - \overline{\mathbf{X}}_i\right)^2}} \,. \tag{1}$$

The correlation sequence $\mathbf{C} = \{c_1, ..., c_I\}$ finally contains the leakage results for all I instructions of the implementation. To check whether an instruction's leakage is significant, we use the standard significance test based on the approximated t-distribution of Pearson's correlation coefficient. The confidence level $(1 - \alpha)$ is set to 99.9999 % and the degrees of freedom df are derived from the number of exponentiations N with $df = (N - 2)$. The significance threshold T is then defined as

$$T = \pm\sqrt{\frac{t_{1-\alpha}^2}{t_{1-\alpha}^2 + df}} \,. \tag{2}$$

All instructions with $c_i \geq T$ or $c_i \leq$ -T are considered to *leak* information about the exponent. In our practical experiments, the tests are done with $N = 5,000$ exponentiations. This number is derived empirically from initial tests of a vulnerable square-and-multiply implementation. In these tests it is sufficient to confidently distinguish leaking and non-leaking instructions. With 5,000 exponentiations, $t_{1-\alpha}$ becomes 4.7590 and the threshold T evaluates to $\pm\,0.0672$.

The advantage of the selected leakage test is that it allows to automatically and efficiently detect leaking code in vulnerable modular exponentiation implementations that have successfully been attacked in literature by observing the execution flow through the instruction cache.

5 Description of the Measurement Framework

The leakage test requires that the execution flow of a program is monitored with instruction-level granularity. In particular the numbers of executions $\mathbf{X}_i$ are needed for each instruction to compute the correlation coefficients. To retrieve

these numbers we observe the instruction pointer (or program counter) over time, because it references all the instructions that are executed during runtime. This is an easy task for instrumentation tools that are normally used for debugging and performance analysis in software development. Instrumenting a compiled program during runtime is called dynamic binary instrumentation (DBI) and typically works by injecting analysis code into the original instruction stream.

For testing programs against instruction cache attacks, dynamic binary instrumentation is beneficial because it allows to monitor a program during runtime without interference from the processor, the operating system, and other applications, which are typical challenges for attacks in practice. DBI also provides greater control over the software under test and supports instruction-level instead of cache-line-level observation granularity. DBI is therefore able to evaluate software under a worst case instruction cache attack with high quality measurements and a cache line size of one instruction. Other, more general benefits are that DBI operates on binaries rather than source code, which might not always be available, especially when third party libraries are used. It avoids the impact of the compiler and its optimizations, because it instruments the actual instructions that are executed and that an adversary would observe during an attack. Compared to static analysis, dynamic instrumentation has the advantage that not all possible code paths must be followed, which quickly gets infeasible to handle for large software components. The analysis effort is reduced to the actual execution flow of a specific input, which is efficient and sufficient for the selected leakage test.

The question of which instrumentation framework to use for the leakage test is not critical. Observing the instruction pointer can be done with any major DBI framework available. We choose Pin [13], which is developed and supported by Intel, because it is a performance-oriented framework that efficiently handles the light-weight instrumentation tasks that we require. This reduces the evaluation time in practice and is especially beneficial for modular exponentiation, which is a costly computation to perform. We implement a custom extension to Pin, a so-called *Pintool*, that attributes the observed instruction pointer addresses to specific instructions within the executed program and its loaded shared libraries.

For the evaluation of the selected cryptographic libraries, we implement a test program that is able to interface and initialize different libraries. It invokes a modular exponentiation operation through the RSA cipher interface provided by each library. In every execution, the test program triggers one RSA decryption with a random exponent and ciphertext. The test program binary is then run by Pin and instrumented by our Pintool, which stores the number of executions of all instructions that were executed at least once. These are the measured $\mathbf{X}_i$ required by the leakage test. Note that the instrumentation is done only for the decryption operation. Library (de-)initialization is omitted, because it adds noise and may consequently prolong the measurement phase. This is achieved by performing the actual decryption in a separate thread and by leaving library setup and teardown to the main program thread. Within the Pintool the instrumentation can then precisely be limited to the decryption thread. In summary, the following steps

are performed to generate the measurements and leakage test results for each RSA implementation:

1. Selection of the implementation under test and compilation of the test program.
2. Execution of the compiled binary with Pin and storage of the instructions' execution counts. This step is repeated N times.
3. Correlation of the exponents' Hamming weights $\mathbf{H}$ with the execution counts $\mathbf{X}_i$.
4. Significance test of the correlation coefficients $\mathbf{C}$ using the threshold T and documentation of the detected leaks.

6 Cryptographic Libraries Under Test

The practical demonstration of the leakage test and the measurement framework is done on an x86-64 system. A selection of nine open-source cryptographic libraries and one textbook implementation of RSA are tested using 2048-bit keys.

Table 1: Overview of the tested cryptographic libraries with corresponding version, usage of CRT components during decryption and the implemented exponentiation algorithm.

Library	Version	CRT	Algorithm
Textbook	-[1]	-	Sqr. & Mul.
wolfSSL	3.8.0	√	Mont. Ladd.
MatrixSSL	3.7.2b	-	Sliding Win.
OpenSSL$_{sw}$	1.0.2g	-	Sliding Win.
LibreSSL$_{sw}$	2.2.5	-	Sliding Win.
Nettle	3.1.1	√	Sliding Win.
mbed TLS	2.2.1	-	Sliding Win.
Libgcrypt	1.6.4	-	Sliding Win.
OpenSSL$_{fw}$	1.0.2g	-	Fixed Win.
LibreSSL$_{fw}$	2.2.5	-	Fixed Win.
BoringSSL	72f7e21[2]	-	Fixed Win.
cryptlib	3.4.2	√	Fixed Win.

Notes:

[1] ..... self-written implementation.
[2] ... shortened git commit hash, because version number not available.

Table 1 displays information about all implementations under test. For each one, the version number (if applicable), the usage of CRT components during decryption (marked with a √) and the fundamental algorithm that is used to

implement modular exponentiation is given. The textbook implementation is a self-written RSA decryption function based on the binary exponentiation algorithm described by Schneier in [25], chapter 11.3. It processes the exponent bit-wise in a square-and-multiply fashion and intentionally exhibits execution flow leaks. We include this implementation for two reasons. First, it is useful to verify our testing approach, which should eventually detect leaking instructions. Second, it serves as a known reference for readers, as square-and-multiply is a common target in previous work. Across the other tested libraries, mainly sliding and fixed window exponentiation algorithms are used. OpenSSL and LibreSSL are listed twice, once with a sliding window (*sw*) and once with a fixed window implementation (*fw*). Fixed window exponentiation is used by default in both libraries. wolfSSL is the only library that implements exponentiation using a Montgomery ladder. From the discussion in Section 3 it is clear that the square-and-multiply and all sliding window algorithms are expected to contain execution flow leaks, whereas the fixed window and Montgomery ladder implementations should not exhibit leaks.

Before the libraries are compiled, most of them automatically adapt themselves to the current system with configuration scripts. In addition, we enable available side-channel countermeasures and other security options, if they are applicable to the RSA decryption functionality. For every library, we implement an interface that allows the test program to (de-)initialize the library, set random exponents and ciphertexts, and trigger decryptions. The majority of the selected libraries performs exponentiation using the private exponent d or provide it as an option. In these cases, a random value is set for d before a decryption is triggered. For all libraries that enforce the use of the CRT components d_p and d_q for exponentiation, a random value is set for d_p while d_q is set to zero. Zeroing the second exponent allows to perform the leakage test identically to the non-CRT case. Although this corrupts the overall decryption result, the first exponentiation is performed normally and the test results of the exponentiation code are not impaired. An alternative approach would be to set random values for both d_p and d_q and to compare the execution counts of the instructions with $\mathrm{HW}(d_p) + \mathrm{HW}(d_q)$. We do not further evaluate this strategy, because setting d_q to zero does not cause any erratic behavior of the libraries in the measurement phase. The choice of CRT or non-CRT exponentiation has no further impact on our tests and where possible we prefer non-CRT exponentiation to simplify interfacing with the cryptographic libraries. More details about the usage of the libraries and all manual configuration steps are provided in Appendix A.

7 Discussion of the Test Results

The output of the leakage test are correlation coefficients for all instructions that are used during RSA decryption. Listing 1.1 shows an excerpt of the test output obtained from MatrixSSL. Every instruction is identified by a hexadecimal offset into the program or shared library binary that is followed by the correlation coefficient.

```
0x0000faf6:  0.5702   (pstm_exptmod:   call   0x5a40)
0x0000fafb:  0.5702   (pstm_exptmod:   test   eax, eax)
0x0000fafd:  0.5702   (pstm_exptmod:   je     0xfaac)
0x0000fbec: -0.9981   (pstm_exptmod:   sub    ax, 1)
0x0000fbf0: -0.9981   (pstm_exptmod:   jne    0xfa8f)
```

Listing 1.1: Excerpt of the test output obtained for MatrixSSL.

If applicable, the corresponding function is given together with the assembly code of the instruction. Every instruction with a correlation coefficient exceeding the significance threshold of 0.0672 is a leak and a potential point of attack for an adversary observing the instruction cache. To compare the cryptographic libraries and the extent of their leakage, we provide three result metrics for each of them. First, the maximum correlation coefficient that is detected over all instructions is given. It indicates the strongest available leak that could potentially be exploited by an adversary. For example, the maximum correlation coefficient detected in the square-and-multiply textbook implementation is 1.0000, indicating a perfect linear relation to the Hamming weight of the private exponent. In contrast, the instructions in the fixed window implementation of cryptlib do not exceed a correlation of 0.0312. Their dependence on the bit values of the private exponent is insignificant in our leakage test. The second and third metric are the total number of leaks and the percentage of leaking instructions of all executed ones. Both indicate how big the attack surface of an implementation is. The more instructions leak, the easier it gets to implement an actual attack in practice.

Table 2: Results of the leakage tests showing the maximum detected correlation coefficient, the number of leaking instructions and their percentage of all executed unique instructions during decryption. Libraries without leaks are highlighted in gray.

Library	**Maximum Corr. Coeff.**	**Leaking Instructions**	**% of Exec. Instructions**
Textbook	1.0000	1642	44.9
wolfSSL	-1.0000	2	0.1
MatrixSSL	-0.9981	5815	52.7
OpenSSL$_{sw}$	0.9907	505	6.4
LibreSSL$_{sw}$	0.9903	522	7.1
Nettle	0.7362	432	8.3
mbed TLS	-0.7061	343	10.1
Libgcrypt	0.5731	676	16.0
OpenSSL$_{fw}$	-0.0497	0	0.0
LibreSSL$_{fw}$	0.0430	0	0.0
BoringSSL	0.0426	0	0.0
cryptlib	0.0312	0	0.0

Table 2 displays the leakage results of all tested implementations. Most importantly it shows that our proposed leakage test effectively detects execution flow leaks and is able to quantify the extent of the leakage. Libraries with sliding window exponentiation, namely MatrixSSL, OpenSSL$_{sw}$, LibreSSL$_{sw}$, Nettle, mbed TLS, and Libgcrypt, exhibit a considerable number of leaks and high maximum correlation coefficients. The risk they face from instruction cache attacks is clearly reflected in the test results. Libraries with fixed window exponentiation, namely OpenSSL$_{fw}$, LibreSSL$_{fw}$, BoringSSL, and cryptlib, have no leaks at all. This reflects their inherent protection against instruction cache attacks as discussed in Section 3. The significant differences between OpenSSL$_{sw}$ and OpenSSL$_{fw}$ as well as LibreSSL$_{sw}$ and LibreSSL$_{fw}$ clearly demonstrate the importance of choosing a protected modular exponentiation implementation. They also show that proper library configuration is vital to the secure usage of cryptographic libraries in practice. Note that the fixed window implementation (*fw*) is enabled by default in both OpenSSL and LibreSSL.

Among the vulnerable sliding window implementations, MatrixSSL has the largest number of leaks comprising over 50 % of its executed instructions. Together with a high maximum correlation coefficient, these results are disturbingly close to the trivial square-and-multiply implementation. Quite different and surprising results are obtained from wolfSSL. Although it implements a Montgomery ladder exponentiation, it exhibits a maximum correlation coefficient of -1.0000. The fact that only two instructions are above the significance threshold strongly suggests that this is a software bug. The source of the two leaks is discussed in more detail in the upcoming Section 8.

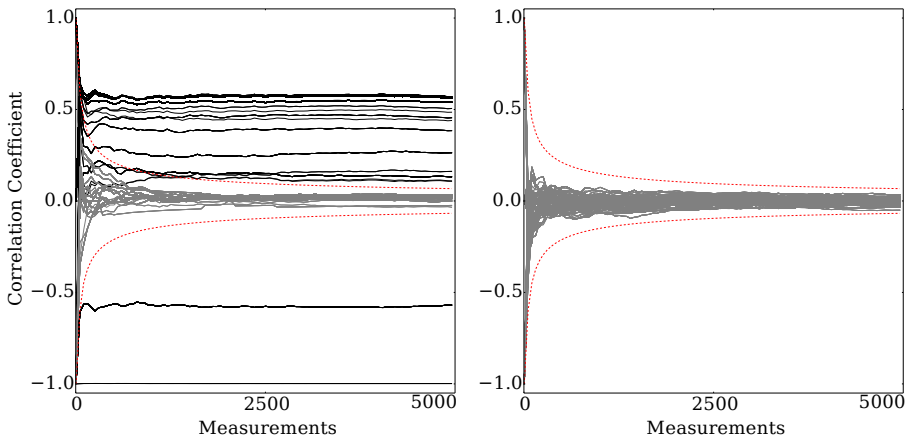

Fig. 1: Correlation coefficients of all executed instructions within MatrixSSL (left) and OpenSSL$_{fw}$ (right) over an increasing number of measurements. The significance threshold is displayed as dotted red lines.

Another illustration of the test results is provided by Figure 1. It displays the correlation coefficients of all executed instructions in MatrixSSL and OpenSSL$_{fw}$ over an increasing number of measurements. The positive and negative significance thresholds are displayed as dotted, red lines. All significant correlation coefficients are plotted in black, all others are colored in gray. The plots clearly visualize the extent of the leakage of the two implementations. The plots also indicate that stable test results are obtained well within 5,000 measurements, which agrees with our initial trials to derive an appropriate number of measurements for the tests. Monitoring 5,000 decryptions on our test system takes on average around 5.50 hours. This is an acceptable time to produce the results for this work and reflects that the execution flow of a single decryption consists of roughly 10^6 to 10^7 executed instructions that must all be instrumented and counted. Note that this time can be significantly reduced by simple optimizations such as using smaller RSA keys, opting for exponentiation with CRT exponents, or parallelizing the measurements. Because each decryption is monitored in a separate Pin process, the parallelization of the measurements can be arbitrary. This shows that the proposed leakage test is not only effective but can also efficiently be adapted to one's computational resources and requirements.

8 Analysis and Patch of wolfSSL

We recommend to patch cryptographic libraries that exhibit execution flow leaks, because these leaks are the basis of instruction cache attacks. Using the test output of wolfSSL, we practically demonstrate how such a patch can look like. To recap, the wolfSSL library implements a Montgomery ladder, which is shown in Algorithm 1. The regular execution flow is achieved with two helper variables R_0 and R_1. The basic idea is to always perform a multiplication and a squaring step, but choose the source and destination helper variables according to the bits of the exponent. Despite implementing this algorithm, two instructions in the wolfSSL library strongly correlate with the Hamming weight of the private exponent. To find the root cause, we start in function `_fp_exptmod` that implements the exponentiation routine. An excerpt of the Montgomery ladder loop body is shown in Listing 1.2. In each loop, the variable `y` contains one bit of the exponent. The multiplication of the helper variables `R[0]` and `R[1]` is done on line 1029 by calling the function `fp_mul`, which is defined in `wolfcrypt/src/tfm.c`. The result is written back to either `R[0]` or `R[1]`, depending on the value of `y`. In `fp_mul`, the call is forwarded to the function `fp_mul_comba`, which is defined in the same source file. The problematic code within `fp_mul_comba`, which was discovered by our test, is shown in Listing 1.3. In general, the function multiplies big integers `A` and `B` and stores the result in `C`. The code in Listing 1.3 checks, whether the reference provided for `C` is equal to `A` or `B`. If this is the case, a temporary variable `tmp` is used to store the intermediate multiplication results. If `C` points to different memory, it is cleared and used to store the intermediate values directly. This is a simple and straightforward optimization to save time and memory.

Input: $g, k = (k_{t-1}, ..., k_0)_2$
Output: $y = g^k$

$R_0 \leftarrow 1;\ R_1 \leftarrow g$
for $j = t - 1$ downto 0 do
  $R_{\neg k_j} \leftarrow R_0 R_1;\ R_{k_j} \leftarrow (R_{k_j})^2$
return R_0

Algorithm 1: Montgomery ladder exponentiation proposed by Joye and Yen [14].

```
1024: /* grab the next msb
..... from the exponent */
1025: y = <next msb>;

1028: /* do ops */
1029: fp_mul(&R[0], &R[1],
..... &R[y^1]); ...
```

Listing 1.2: Code excerpt from the Montgomery ladder exponentiation in `_fp_exptmod` (wolfSSL).

```
453: if (A == C || B == C) {
454:    fp_init(&tmp);
455:    dst = &tmp;
456: } else {
457:    fp_zero(C);
458:    dst = C;
459: }
```

Listing 1.3: Leaking code in the `fp_mul_comba` multiplication routine (wolfSSL).

```
453:
454: // fix: always use
455: // temporary variable
456:
457: fp_init(&tmp);
458: dst = &tmp;
459:
```

Listing 1.4: Replacement for the leaking code in the multiplication routine shown in Listing 1.3.

The problem is that when `fp_mul_comba` is called from within the Montgomery ladder, the variable `C` contains either the reference to `R[0]` or `R[1]`, depending on the exponent. If the current exponent bit is 1, `C` contains the reference to `A`. In this case, the `if` clause on line 453 in Listing 1.3 only evaluates the first condition `(A == C)`, which is true and sufficient to enter the conditional code. If the current exponent bit is 0, `C` contains the reference to `B`. In this case, the `if` clause evaluates both conditions `(A == C)` and `(B == C)` until it enters the conditional code. In other words, the condition `(B == C)` is only evaluated, if the current exponent bit is 0. The instructions checking this second condition are exactly those two that correlate (negatively) with the Hamming weight of the private exponent (see Table 2).

A simple fix of this problem is given in Listing 1.4. It removes the `if` clause and always uses the temporary variable `tmp`. To verify the effectiveness of this fix, we repeated the leakage test for the patched wolfSSL library. The results are shown in Table 3. The patched version is completely free from leaks. Note that the maximum correlation coefficient is now less than 0.0000, which means that the instructions during decryption are either executed a constant number of times or simply do not exhibit a linear relation with the Hamming weight of the private exponent. Since the `if` clause was introduced to improve performance, we also tested the decryption speed of the patched version. As shown in Table 3, the patch decreases performance by 0.05 %, which is a negligible slowdown.

In the spirit of responsible disclosure, wolfSSL was informed by the authors about the discovered leaks prior to publication of this paper.

Table 3: Comparison of the original and patched wolfSSL library. The speed comparison is based on 10^7 decryptions.

Library	Maximum Corr. Coeff.	Leaking Instructions	Speed (decr. / s)	Speed (relative)
original	-1.0000	2	129.87	1.0000
patched	< 0.0000	0	129.81	0.9995

9 Conclusion

The instruction cache is an established side-channel that allows an adversary to learn applications' secret data that is leaked through their execution flow. Numerous publications and practical attacks have been demonstrated in the past decade, many of which focus on implementations of modular exponentiation. Although robust algorithms for modular exponentiation are available, various cryptographic libraries use implementations that are inherently vulnerable to instruction cache attacks. To test cryptographic software for instruction cache leaks, we propose an effective leakage test that can be efficiently conducted with readily available dynamic binary instrumentation tools. With this test we detect and quantify leaks in multiple cryptographic libraries with instruction-level granularity. Since instruction cache leaks have been shown to be a threat in practice, we recommend libraries to carefully select the algorithm for modular exponentiation. Square-and-multiply-always, Montgomery ladder, and fixed window exponentiation are all practicable choices that limit the leakage an adversary can exploit via the instruction cache. Furthermore, we recommend testing any final library binary to ensure that no additional leaks are introduced in the software development process. The analysis and fix of a non-trivial execution flow leak in wolfSSL clearly demonstrates the necessity of this final check. For the future, the proposed leakage test may be applied to operations similar to modular exponentiation, e.g. scalar multiplication in elliptic-curve based schemes. Further work might also establish a systematic link between a detected execution flow leak and the number of exponent bits an adversary can learn from observing a given implementation via the instruction cache. For now, we consider every significant leak as a potential point of attack and therefore recommend to remove all of them.

A Configuration of Cryptographic Libraries

Most of the cryptographic libraries configure themselves with automatic scripts that adapt the library to the current processor, which is an Intel Xeon E5440 in our tests. The paragraphs beneath discuss all manual configuration steps and also list the functions and source code files containing the exponentiation code. This is important to verify and reproduce our results, because libraries often feature multiple implementations of modular exponentiation and pick one depending on their configuration.

wolfSSL On our x86-64 test system, wolfSSL automatically speeds up public key operations by defining the `fastmath` option. In addition, we define the flag `TFM_TIMING_RESISTANT`, because it is recommended in the documentation [29]. The exponentiation function `_fp_exptmod` is defined in `wolfcrypt/src/tfm.c`.

MatrixSSL For MatrixSSL no extra configuration steps are done. The exponentiation function `pstm_exptmod` is implemented in `crypto/math/pstm.c`.

Nettle This library offers two functions for RSA decryption, `rsa_decrypt` and `rsa_decrypt_tr`. We use the latter one, because it implements base blinding to mitigate timing attacks and is therefore recommended in practice. For the required random numbers, we use the included implementation of the Yarrow PRNG. By default Nettle calls the external GNU multi precision arithmetic library (GMP) [26] to perform exponentiation. In particular, it relies on the function `mpn_powm` in the source file `mpn/generic/powm.c`. We compile GMP in version 6.1.0 with default configuration and link it with decryption program.

mbed TLS The mbed TLS library is compiled with the flag `MBEDTLS_RSA_NO_CRT` to enforce decryption without CRT components. Setting this flag does not affect the exponentiation code in function `mbedtls_mpi_exp_mod` defined in `library/bignum.c`. For the implemented base blinding countermeasure the `CTR_DRBG` random number generator is used.

Libgcrypt The Libgcrypt library implements a secure memory feature that can be used for key storage. Because this is not required in our tests and does not change the exponentiation algorithm, we disable the feature during runtime. Exponentiation is done in function `_gcry_mpi_powm`, which is implemented in `mpi/mpi-pow.c`.

cryptlib Before triggering the decryption in the cryptlib library, we set the `CRYPT_RANDOM_SLOWPOLL` and `CRYPT_OPTION_MISC_SIDECHANNELPROTECTION` options. A random slowpoll gathers entropy before the library actually needs it, which speeds up the retrieval of random numbers at a later point in time. The enabled side-channel protection activates a base blinding countermeasure, sanity checks in the code and the constant-time, fixed window exponentiation in function `BN_mod_exp_mont_consttime`, which is defined in `bn/bn_exp.c`.

OpenSSL and Forks The OpenSSL library is compiled with the options `OPENSSL_BN_ASM_MONT` and `OPENSSL_BN_ASM_MONT5`, which activate assembly optimized Montgomery multiplication. By default, the base blinding countermeasure and the constant-time, fixed window exponentiation in function `BN_mod_exp_mont_consttime` defined in `crypto/bn/bn_exp.c` are enabled. The LibreSSL and BoringSSL libraries are compiled and used with the default options, which enable a similar exponentiation implementation as in OpenSSL. Sliding window exponentiation can be manually enabled in OpenSSL and LibreSSL by setting the `RSA_FLAG_NO_CONSTTIME` flag during runtime.

References

1. Acıiçmez, O.: Yet another microarchitectural attack: Exploiting i-cache. In: Proceedings of the 2007 ACM Workshop on Computer Security Architecture. pp. 11–18. CSAW '07, ACM (2007)
2. Acıiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010, Lecture Notes in Computer Science, vol. 6225, pp. 110–124. Springer Berlin Heidelberg (2010)
3. ARM Limited: mbed TLS (2016), https://tls.mbed.org/
4. Barthe, G., Betarte, G., Campo, J., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1267–1279. CCS '14, ACM, New York, NY, USA (2014)
5. Chen, C., Wang, T., Kou, Y., Chen, X., Li, X.: Improvement of trace-driven i-cache timing attack on the rsa algorithm. J. Syst. Softw. 86(1), 100–107 (Jan 2013)
6. Coron, J.S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, c.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, vol. 1717, pp. 292–302. Springer Berlin Heidelberg (1999)
7. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). pp. 431–446. USENIX, Washington, D.C. (2013)
8. Doychev, G., Köpf, B.: Rigorous analysis of software countermeasures against cache attacks. https://arxiv.org/abs/1603.02187v2 (2016)
9. Google Inc.: boringssl (2016), https://boringssl.googlesource.com/boringssl/
10. Gordon, D.M.: A survey of fast exponentiation methods. J. Algorithms 27(1), 129–146 (Apr 1998)
11. Gutmann, P.: cryptlib (2016), https://www.cs.auckland.ac.nz/~pgut001/cryptlib/
12. INSIDE Secure Corporation: MatrixSSL (2016), http://www.matrixssl.org
13. Intel Corporation: Pin - A Dynamic Binary Instrumentation Tool (June 2012), https://software.intel.com/en-us/articles/pintool
14. Joye, M., Yen, S.M.: The montgomery powering ladder. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 291–302. Springer (2002)
15. Knuth, D.: The Art of Computer Programming: Seminumerical algorithms. Addison-Wesley series in computer science and information processing, Addison-Wesley (1981)
16. Koç, C.K.: Analysis of sliding window techniques for exponentiation. Computers and Mathematics with Applications 30, 17–24 (1995)
17. Koch, W.: Libgcrypt (2016), https://www.gnu.org/software/libgcrypt/
18. Langley, A.: ctgrind - checking that functions are constant time with valgrind (2010), https://github.com/agl/ctgrind
19. Möller, N.: Nettle - a low-level cryptographic library (2016), https://www.lysator.liu.se/~nisse/nettle/
20. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. Cryptology ePrint Archive, Report 2005/368 (2005), http://eprint.iacr.org/2005/368

21. OpenBSD: LibreSSL (2016), http://www.libressl.org/
22. Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005 (2005)
23. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? Cryptology ePrint Archive, Report 2016/1123 (2016), http://eprint.iacr.org/2016/1123
24. Rodrigues, B., Quintão Pereira, F.M., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 110–120. CC 2016, ACM, New York, NY, USA (2016)
25. Schneier, B.: Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C. John Wiley & Sons, Inc., New York, NY, USA (1995)
26. The GNU project: The GNU Multiple Precision Arithmetic Library (2016), https://gmplib.org/
27. The OpenSSL Project: Changes between 0.9.7g and 0.9.7h [11 Oct 2005] (October 2005), https://www.openssl.org/news/changelog.html
28. The OpenSSL Project: OpenSSL (2016), https://www.openssl.org/
29. wolfSSL: wolfSSL User Manual (March 2016), https://www.wolfssl.com/documentation/wolfSSL-Manual.pdf, v3.9.0
30. wolfSSL Inc. : wolfSSL Embedded SSL Library (2016), https://www.wolfssl.com
31. Yarom, Y., Falkner, K.: Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014. pp. 719–732 (2013)